Core Java: Functional Interfaces (java.util.function)

Persistent Interactive | Persistent University

Objectives :

At the end of this module, you will be able to understand:

- Functional Interfaces
- Built-in interfaces included in java.util.function
- Core Interfaces Predicate, Function, Consumer and Supplier
- Implementing core interfaces
- Implementing primitive and binary variations of the base interfaces of the java.util.function package

Functional Interfaces

- Any interface with a *SAM(Single Abstract Method)* is a functional interface, and its implementation may be treated as lambda expressions.
- *Functional interfaces* provide target types for lambda expressions and method references.
- Each functional interface has a single abstract method, called the *functional method* for that functional interface, to which the lambda expression's parameter and return types are matched or adapted.

Functional Interfaces

- Functional interfaces can provide a target type in multiple contexts, such as assignment context, method invocation, or cast context:

```
// Assignment context
Predicate<String> p = String::isEmpty;

// Method invocation context
stream.filter(e -> e.getSize() > 10)...

// Cast context
stream.map((ToIntFunction) e -> e.getSize()).
```

Why java.util.function?

- There are a lot of re-usable functional requirements that can be captured by functional interfaces and lambdas.
- The designers of Java 8 have captured the common use cases and created a library of functions for them.
- A new package called java.util.function is hosting these common functions.

Some Interfaces from In-Built functions library

- Function
  - Represents a function that accepts one argument and produces a result.
- Consumer
  - Represents an operation that accepts a single input argument and returns no result.
- Predicate
  - Represents a predicate (boolean-valued function) of one argument.
- Supplier
  - Represents a supplier of results.

Some Interfaces from In-Built functions library

- BiFunction
  - Represents a function that accepts two argument and produces a result.
- BiConsumer
  - Represents an operation that accepts two input argument and returns no result.
- BiPredicate
  - Represents a predicate (boolean-valued function) of two arguments.

Implementing core interfaces using Lambda

```
Function<Integer, Integer> getSquare=(no)-> {return
(no*no);};

BiFunction<Integer, Integer, String> add=(a,b)->"
addition= "+(a+b);

Predicate<Integer> isEven=(no)->no%2==0;

BiPredicate<Integer, Integer>
isEvenAndDivisible=(no,divisor)-> no%2==0 &&
no%divisor==0;
```

Implementing core interfaces using Lambda

```
Supplier<String> messageSupplier=()->" Good Morning!"

Consumer<String> consumer=(a)->{System.out.println(a);};

BiConsumer<String, Integer> biConsumer=(name,marks) -> {System.out.println("name :"+name+" marks:"+marks);};
```

Problems with prior java versions?

- Java Type can be
  - a primitive type (int, double, byte, char)
  - a reference type (String, Integer, Object, List)
- Generic parameters can be bound only to reference types as per internal generics implementation
- AutoBoxing convert a primitive type into a corresponding reference type and vice versa.
- Boxed values use more memory and require additional memory lookups to fetch the wrapped primitive value and adds more performance cost.
- Java 8 brings a specialized version of the functional interfaces in order to avoid autoboxing operations when the inputs or outputs are primitives.

Primitive and Binary variations solves this

- Java 8 brings a specialized version of the functional interfaces in order to avoid autoboxing operations when the inputs or outputs are primitives.

```
@FunctionalInterface
public interface IntPredicate {
public boolean test(int i); ...
}

// IntPredicate avoids a boxing operation of the value 1974
IntPredicate ip = i -> i == 1974;
ip.test(1974);

//whereas using a Predicate<Integer> would box the argument 1974 to an Integer object
Predicate<Integer> p = i -> i == 1974;
p.test(1974);
```

Summary

With this we have come to an end of our session, where we discussed about

- Functional interfaces
- Built in functional interfaces
- Implementing core interfaces using Lambda

Appendix

References

Thank you

Reference Material : Websites & Blogs

https://docs.oracle.com/javase/8/docs/api/?java/util/function/package-summary.html

Reference Material: Books

Java SE 8 for the Really Impatient

By: Cay S Horstmann

Java 8 Lambaddas

By: Richard Warburton

Persistent

Thank you!

Persistent Interactive | Persistent University